File Sharing On Linux

by Primoz Gabrijelcic

This started as a simple sequel to my article on using the file system for interprocess synchronisation (*Let's Cooperate*, in Issue 68). I planned to take my supposedly well-written and cross-platform-ready code, port it to Kylix, and write a short article about the (again supposedly) simple task. Yeah, sure, you probably all know how such well-thought-out plans end. I've now been wrestling with Linux and file synchronisation (not with Kylix, since that is a nice quiet animal) for more than two months and I still don't think that I know half the tricks there are. But at least I have managed to achieve my goal: I can present you the code that synchronises processes on both Windows and Linux platforms using only the file system.

What Works On Windows...

To better understand the problems of porting the synchronisation code to Linux, let's first take a short tour of the above-mentioned article. First a short note for newcomers: that article presented a family of process-synchronisation primitives, all based on the file system. There was an inter-process (and inter-computer) mutex that used one shared file to accomplish the task, a critical section, a group (or pool), and even more complicated synchronisation and communication classes. (If you want to learn more, read the article.)

The main topic of this article is the simplest synchronisation primitive: a mutex. On Windows, implementation of a mutex is simple. I represented it with a single file. The process opened that file with exclusive access to acquire the mutex. To release the mutex, the process closed the file.

That approach works well with misbehaving applications and hardware. If the process terminates without releasing the mutex (closing the file), the operating system will do it instead. If things go from bad to worse and the computer running that process crashes (while the process has ownership of the mutex), the file server will notice that (sooner or later) and will release the file so that it can be acquired by another program.

...May Not Work On Linux

So what is wrong with Linux? Well, nothing, actually. The problem is more of a philosophical nature. In the Windows world we are used to file locking. It is something natural, something that permeates the operating system, something that we had to learn when we stopped using DOS. But for all practical purposes, Linux is Unix, and Unix was designed with different things in mind, one of which was openness and sharing. From the very old days, Unix has supported different kinds of file systems, including those that span a network. Remember, that was well before Windows, and file locking was something that was simply too expensive (in terms of time and network traffic) to be implemented. And that is why *there is no locking on Linux*.

Take this with a grain of salt: I'm telling the truth, but not the complete truth. There is locking on Linux (and Unix), but it is not *mandatory* (as on Windows) but *advisory*. In other words, on Windows you can *forbid* other files to access the file you are working on; on Linux you can only *notify* well-behaved programs that you are working on the file and can they please come back later.

Even that is not completely true: there is mandatory locking on Unix (and Linux), it is just not widespread. More on that later.

When I found that horrifying revelation, I was stunned (maybe you already noticed that I'm an old Windows geezer, not a Linux guru?). I just had to find out how they do it on the other side of the fence. After much (and I mean *much*) surfing, during which I found lots of interesting links (all neatly collected in the sidebar), the picture became clearer.

Most Linux applications use the trivial approach: lock files. If the file exists, the resource is locked (the mutex is acquired). If the file doesn't exist, the resource is free. If the process crashed before it could release the file, the resource is locked and the situation is called a feature. Really. To quote a beautiful *Secure Programming for Linux and Unix HOWTO* (section *Avoid Race Conditions*): 'On Unix-like systems resource locking has traditionally been done by creating a file to indicate a lock, because this is very portable. It also makes it easy to "fix" stuck locks, because an administrator can just look at the file system to see what locks have been set. Stuck locks can occur because the program failed to clean up after itself (eg, it crashed or malfunctioned) or because the whole system crashed. Note that these are "advisory" (not "mandatory") locks; all processes needing the resource must cooperate to use these locks.'

OK, I can buy the *portability* argument but all that *easy to fix* part just describes a way of fixing the broken system. I for sure wouldn't expect users of my program to delete the stuck locks when the program crashes. Would you?

Another problem with this approach is that it may not always work. Remember, Linux supports plenty of file systems and some of them are not local but networked (the files are on another computer, not on the one we are working on). On Windows, this situation is typically hidden from us as Windows tries hard to represent the network file system as if it were local. On Linux, this may not be the case. For example, the most common network file system, NFS, doesn't

completely support normal file open/create semantics in version 2. (Translation: when using NFS 2, open and create may behave differently to the open and create on the local file system). Because of that, using lock files for synchronisation may get you into a race condition.

There are solutions for that, too (the best is to use NFS version 3 instead of version 2), but it also shows how much one file system may differ from another on Unix.

A File Is A Mutex

Let's first try to implement the simple approach from Windows: a lock file that signals the *acquired* state of the mutex with its own presence. This may seem trivial at first: to acquire the mutex, we will `Assign` a file and call `Rewrite`; to release it, we will `Close` and delete the file. Alas, this will not work. `Rewrite` allows a process to overwrite a file owned by another process and that breaks our scheme.

The reason lies deep inside the `System` unit in the `OpenFile` function. It uses a function from the system library, `libc.open`, to open files. This is fine, but the combination of flags sent to the `open` from `Rewrite` is not fine, at least for our purposes. Kylix sets the rewrite flags to `O_CREAT or O_TRUNC or O_RDWR`, which in translation means: create the file if it doesn't exist, truncate the file if it does exist, and open it for reading and writing. Because there is no locking, Linux allows you to do that from more than one process at the same time, and that's why two processes can simultaneously `Rewrite` the lock file both thinking that they own the mutex. Not good.

To test this, we can write a small Kylix program (shown in Listing 1) that rewrites the file, waits for `Enter` to be pressed, and then closes the file. Compile the program, open a console window, run `./testrewrite` and let it wait at the point where it waits for the `Enter` key. Then open another console window and run `./testrewrite` again. You'll see that the second instance is also able to rewrite the file without a problem.

```
program testrewrite;
{$APPTYPE CONSOLE}
var
  f: file;
begin
  Assign(f,'rewrite-test');
  Writeln('Will rewrite file...');
  Rewrite(f);
  Writeln('...rewritten');
  Writeln('Press Enter to close the file');
  Readln;
  Close(f);
end.
```

➤ *Listing 1: Simultaneous write access testing.*

The correct way to open a file for exclusive access (if we ignore the NFS 2 problem I mentioned before) is to specify another combination of flags in the `open` call. Instead of the `O_TRUNC` we must specify `O_EXCL`, which makes sure that the file is only created if it doesn't exist. Instead of `O_RDWR` we will use the more restrictive `O_WRONLY` (because we will not read from the file at all). To do this, we must call the `open` function directly:

```
open(fileName, O_CREAT or
  O_EXCL or O_WRONLY, 0)
```

If `open` fails to create the file, it will return `-1`. In that case, the `GetLastError` function will tell us whether the `open` failed because the target file already exists (the error will be `EEXIST`) or for some other reason. To implement the `Acquire` as we did months ago for Windows, we'll have to loop until the file is successfully created or until the allowed time (specified by the caller) is exceeded (see Listing 2).

At least releasing the mutex is trivial: we only have to close the handle and delete the lock file (the `Release` method is shown in Listing 3). Actually, the handle to the lock file is not really needed and it would be possible to close the handle in the `Acquire` method immediately after the file is successfully created. `TGpFileExistsMutex` keeps the handle open only because it can then be used to test if the mutex is currently acquired or not (the `Acquired` function

➤ *Listing 2: Acquiring a file-based mutex.*

```
function TGpFileExistsMutex.Acquire(timeout: DWORD): boolean;
var
  err  : TErrorVal;
  start: int64;
begin
  if Acquired then
    raise EGpFileSync.CreateFmt(SAlreadyAcquired,[SyncFile])
  else begin
    start := GetTickCount;
    repeat
      fmHandle := open(PChar(SyncFile),O_WRONLY OR O_CREAT OR O_EXCL, 0);
      if fmHandle = INVALID_HANDLE_VALUE then begin
        err := GetLastError;
        if err = EEXIST then begin
          if not Elapsed(start,timeout) then
            Sleep(RetryDelay)
        end else
          raise EGpFileSync.CreateFmt(SCannotAccessFile,[
            SyncFile,SysErrorMessage(err)]);
      end else
        err := 0;
    until (err = 0) or Elapsed(start,timeout);
    Result := (err = 0);
  end;
end; { TGpFileExistsMutex.Acquire }
```

➤ *Listing 3: Releasing a file-based mutex.*

```
procedure TGpFileExistsMutex.Release;
begin
  if Acquired then begin
    __close(fmHandle);
    DeleteFile(SyncFile);
    fmHandle := INVALID_HANDLE_VALUE;
  end else
    raise EGpFileSync.CreateFmt(SNotAcquired,[SyncFile]);
end; { TGpFileExistsMutex.Release }
```

```
function TGpLinkedFileMutex.Acquire(timeout: DWORD):
  boolean;
var
  err  : TErrorVal;
  start: int64;
  selfName: array [0..255] of char;
  fileStat: TStatBuf;
  gotLock: boolean;
begin
  if Acquired then
    raise EGpFileSync.CreateFmt(SAlreadyAcquired,[SyncFile])
  else begin
    gotLock := false;
    gethostname(selfName,SizeOf(selfName)-1);
    fmUniqueFile := Format('%s%s:%d:%d',
      [ExtractFilePath(ExpandFileName(SyncFile)),
      selfName,getpid, GetCurrentThreadID]);
    fmHandle := open(PChar(fmUniqueFile),O_WRONLY OR
      O_CREAT OR O_EXCL, 0);
    if fmHandle = INVALID_HANDLE_VALUE then begin
      err := GetLastError;
      raise EGpFileSync.CreateFmt(SCannotAccessFile,[
        fmUniqueFile,SysErrorMessage(err)]);
    end else begin
      start := GetTickCount;
      repeat
        if link(PChar(fmUniqueFile),PChar(SyncFile)) = 0
          then
          if stat(PChar(fmUniqueFile),fileStat) = 0 then
            gotLock := fileStat.st_nlink = 2;
      until gotLock or Elapsed(start,timeout);
    end;
    if gotLock then
      Result := true
    else begin
      __close(fmHandle);
      fmHandle := INVALID_HANDLE_VALUE;
      unlink(PChar(fmUniqueFile));
      Result := false;
    end;
  end;
end; { TGpLinkedFileMutex.Acquire }
```

➤ *Listing 4: Acquiring a linked file mutex.*

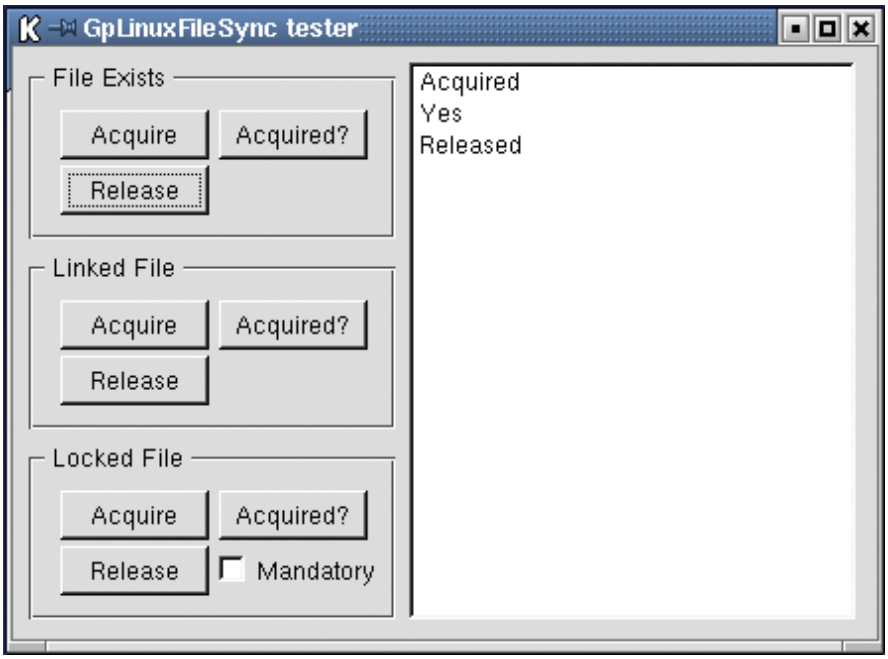
simply tests the `fmHandle` against `INVALID_HANDLE_VALUE`).

All the Linux mutex implementations I'll show you are stored in the file GpLinuxFileSync. They share the common ancestor class, `TGpFileSynchroObject`, that defines some common behaviour. To test the classes, I have written a simple test program, shown in Figure 1 and available on this month's disk. To test any of the mutex implementations, run two copies of the program from the same directory and then play with the `Acquire` and `Release` buttons.

Linked Files

To solve the NFS version 2 problem, we need another approach.

➤ *Figure 1*




```
procedure TGpLinkedFileMutex.Release;
begin
  if Acquired then begin
    __close(fmHandle);
    fmHandle := INVALID_HANDLE_VALUE;
    unlink(PChar(fmUniqueFile));
    unlink(PChar(SyncFile));
  end else
    raise EGpFileSync.CreateFmt(SNotAcquired,[SyncFile]);
end; { TGpLinkedFileMutex.Release }
```

➤ *Listing 5: Releasing a linked file mutex.*

Instead of using file creation to guarantee atomicity, we will be using file linking.

Linking is a feature of all Unix file systems. Two different concepts are hidden under a common name: hard linking and soft linking. The former is a way to give the physical file two different names. Both names point to the same data and one is indistinguishable from the other *[See the following article on NTFS hard links to see how Windows does it. Ed]*. The latter, which interests us more, is just a pointer to another file or directory, similar to a Windows shortcut, only implemented on the lower (file system) level. To the application, the symbolic link appears as a normal file or folder.

The manpage for `open(2)` recommends the following approach to creating lock files on all file systems. Create a unique file on the file system where the lock file is stored. Use the `link` call to make the link to the lock file from the unique file. Ignore the result of the `link` call. Use the `stat` call on the unique file to check if its link count has increased to 2. You can read manpages online, by the way:

```
www.linuxcentral.com/linux/
  man-pages/open.2.html
```

Instead of file creation this approach uses the `link` system function to guarantee atomicity. Even on NFS volumes, `link` guarantees that only one process is able to link to the lock file, even if two processes start the linking operation at precisely the same time. The `stat` step must be used because `link` may not return the correct result on NFS.

Although the approach looks quite convoluted, it is really simple

to implement. To create a unique file we can use the name of the computer (retrieved with the `gethostname` call) and the process ID (retrieved with the `getpid` call). To be completely sure of uniqueness, we can also add the ID of the current thread. Kylix kindly provides the portable `GetCurrentThreadID` function to do the latter. The `Acquire` method implemented in this manner is shown in Listing 4.

Releasing the mutex is almost as simple as in the previous version: we only have to close, unlink and delete the unique file (Listing 5).

Advisory Locking

Remember what I have said at the beginning: there is no locking on Linux? That was, well, a lie. Or at least an oversimplification. It would be more correct to say that there is no *mandatory* locking on Linux. If applications are working together, they can achieve near-locking nirvana even on Linux.

Unix versions (and Linux with them) typically support a notion of *advisory* locking. This is a concept that is strange to Windows-only programmers: it dictates that an application should call a special locking function before accessing a shared file. The function will either lock the file and allow the application to proceed, or return an error.

It should be said loud and clear that a file locked in this manner remains locked only for applications that are using advisory locking to access it. If a misbehaved application tries to directly access an advisory-locked file, *it will succeed*! This is the true Unix spirit: live by the rules and you will peacefully coexist with others. If that doesn't work, bypass the rules and do it your own way (and take the blame when others have problems because of you).

What are the problems? Besides misbehaved applications, there are plenty. Some file systems may not support advisory locking. You'll have a hard time configuring NFS 3 to support it. Forget NFS 2, it doesn't support advisory locking at all. And you definitely won't be able to use advisory locking for cross-platform work. At least nobody could help me configure my Samba installation in a way that will make Windows locking and Linux advisory locking peacefully coexist on an SMB mount. (A plea for help: if you do know how to make this work, contact me!)

On the other hand, some good words can be said on behalf of advisory locking (at least when used via the `fcntl` call, the way we will use it). First, it is POSIX-compliant. In short, that means that it is implemented in the same way on most Unix systems. Second, it should work with a properly-configured NFS file system (if you are interested, in this case `fcntl` requests are caught by the `rpc.lockd` daemon, which forwards them to the `lockd` on the server host). Third, it can lock only part of the file and can differentiate between read and write locks. Lastly, if a process dies, its locks are automatically removed.

Let's go back to the API. To advisory lock a file, we can use the `fcntl` function (there is another way to lock a file, an `flock` call, but it doesn't work with NFS). But before we can use `fcntl`, we need a file handle: writeable if we will write-lock it, but for read-locking read access will do.

Then we'll call the `fnctl` function. It accepts three parameters: the handle of the file we will be working with, a command describing the operation we want to execute and an argument to this command. `Fcntl` can work in many ways (governed by the command parameter) from duplicating the file descriptor to retrieving information on its owner. Somewhere in between, it can also lock the file.

To do this, we must pass `FD_SETLK` as the second parameter

➤ *Listing 6: Locking a file for reading or writing.*

```
function LockFile(FileHandle: integer; forReading: boolean): integer;
var
  LockVar: TFLock;
begin
  with LockVar do begin
    l_whence := SEEK_SET;
    l_start := 0;
    l_len := 0;
    if forReading then
      l_type := F_RDLCK
    else
      l_type := F_WRLCK;
  end;
  Result := fcntl(FileHandle, F_SETLK, LockVar);
end; { LockFile }
```

➤ *Listing 7: Acquiring a locked file mutex.*

```
function TGpLockedFileMutex.Acquire(timeout: DWORD):
  boolean;
var
  err  : TErrorVal;
  start: int64;
  mode: cardinal;
begin
  if Acquired then
    raise EGpFileSync.CreateFmt(SAlreadyAcquired,[SyncFile])
  else begin
    mode := S_IRUSR OR S_IWUSR OR S_IRGRP OR S_IWGRP
      OR S_IROTH OR S_IWOTH;
    if fmMandatory then
      mode := mode OR S_ISGID;
    fmHandle :=
      open(PChar(SyncFile),O_WRONLY + O_CREAT, mode);
    if fmHandle = -1 then
      raise EGpFileSync.CreateFmt(SCannotAccessFile,[
        SyncFile,IntToStr(errno)]);
    err := 0;
    try
      start := GetTickCount;
      repeat
        err := LockFile(fmHandle,false);
        if err = -1 then begin
          if (errno = EAGAIN) or (errno = EACCES) then begin
            if not Elapsed(start,timeout) then
              usleep(RetryDelay*1000)
          end else
            raise EGpFileSync.CreateFmt(SCannotAccessFile,[
              SyncFile,IntToStr(errno)])
        end;
      until (err = 0) or Elapsed(start,timeout);
    finally
      if err <> 0 then begin
        __close(fmHandle);
        fmHandle := INVALID_HANDLE_VALUE;
      end;
    end;
    Result := (err = 0);
  end;
end; { TGpLockedFileMutex.Acquire }
```

and the address of the structure `TFLock` (declared in the unit `Libc`). A helper function `LockFile` (see Listing 6) will set this structure to lock the whole file for reading or writing (depending on a parameter), call the `fcntl` function and return the status code.

If `LockFile` returns `0`, we have the lock and we can proceed. Otherwise, we'll wait a little and retry, as in all the implementations we've seen so far. To release the lock, we only have to close the file handle. `Acquire` is somewhat overcomplicated, because it also tries to implement mandatory System V locking (described next): it is shown in Listing 7.

As you may have noticed, the `TGpLockedFileMutex.Acquire` routine checks the internal flag named `fmMandatory` before opening the file. What is that? Didn't I just tell you that Linux only knows about advisory locking?

Well, that was again a slight misinformation. There is a way to mandatory-lock a file: it is called System V mandatory locking. System V is of course the brand of Unix where this locking was first implemented. It is rarely used, usually only works internally on one computer (so forget about multi-computer and multi-platform synchronization), and cannot help us in synchronizing Windows and Linux applications. Nevertheless, I have added the mandatory locking support to the `TGpLockedFileMutex` class (using the parameter `mandatoryLock` in the constructor) if somebody wants to play with it.

Where To Keep The Lock

An important question is where the lock file should be stored. Of course, if you'll be synchronizing Windows and Linux machines, you'll put the lock file on the SMB mount. But if you only want to lock a resource on a local machine, you have many options.

My advice would be to adhere to the *Filesystem Hierarchy Standard* (FHS: www.pathname.com/fhs/), a document that tries to bring some order in the typical Unix mess. FHS advises on the best placement of more important classes of files (shared programs, system utilities, application settings, lock files...).

If you just want to be sure that your application doesn't execute more than once on a given machine, the FHS advice is to create a lock file /var/run/NAME.pid where NAME is the application name and pid is its process ID, which the `getpid` function will kindly provide. If you are synchronizing access to a device, you should create this file in /var/lock. For some other ideas, check Section 2 of the FHS.

Conclusions

As you have seen, synchronizing Linux processes is hard but not impossible with some careful programming. But I should also add that file-based solutions are really useful only to synchronize access to rarely used resources (for example, shared files that are accessed less than once every few seconds). If your requirements are higher, you should search for another solution. A database, maybe, or a custom synchronization server using TCP/IP.

Linux Programming Resources

Secure Programming for Linux and Unix HOWTO / Structure Program Internals and Approach / Avoid Race Conditions:
www.linuxdocs.org/HOWTOs/Secure-Programs-HOWTO/avoid-race.html

Advanced Linux Programming:
www.advancedlinuxprogramming.com/advanced-linux-programming.pdf

HERT Tutorial Links: www.hert.org/docs/tutorials/

Programming Texts and Tutorials:
http://stommel.tamu.edu/~baum/programming.html

Unix Programming FAQ: www.landfield.com/faqs/unix-faq/programmer/faq/

Secure Programming for Linux and Unix HOWTO:
www.dwheeler.com/secure-programs/

Linux programming: www.linuxprogramming.com/

Mandatory File Locking For The Linux Operating System:
www.linuxhq.com/kernel/v2.0/doc/mandatory.txt.html

Linux Central Man Pages: www.linuxcentral.com/linux/man-pages/

Linux MAN Pages Indexed HTML version: http://linux.ctyme.com/

Linoleum Linux programming resources: http://leapster.org/linoleum/

The LinuX files: www.cplus.about.com/compute/cplus/cs/thelinuxfiles/

Filesystem Hierarchy Standard: www.pathname.com/fhs/

Primoz Gabrijelcic is the R&D Manager of FAB d.o.o. in Slovenia. You can contact him at gp@fab-online.com